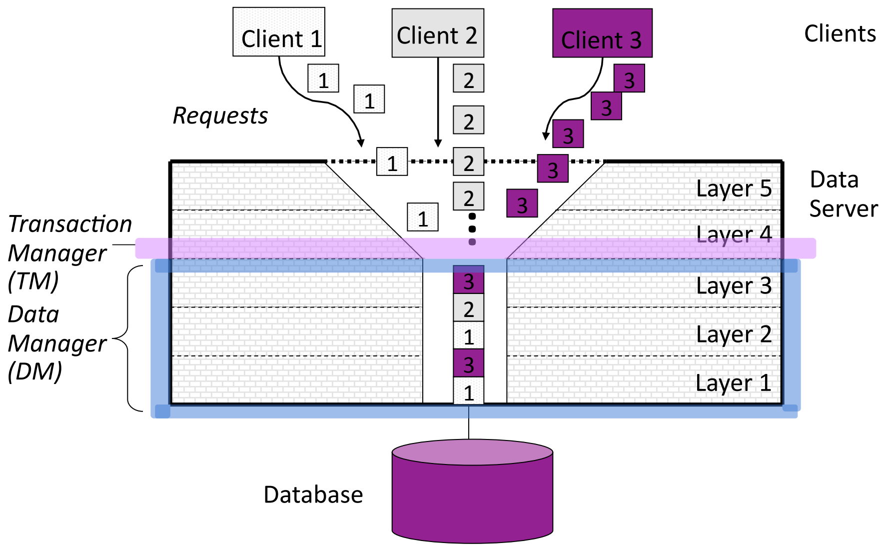
CMPSC 274: Transaction Processing Lecture #4: Concurrency Control Protocols

Divy Agrawal
Department of Computer Science
UC Santa Barbara

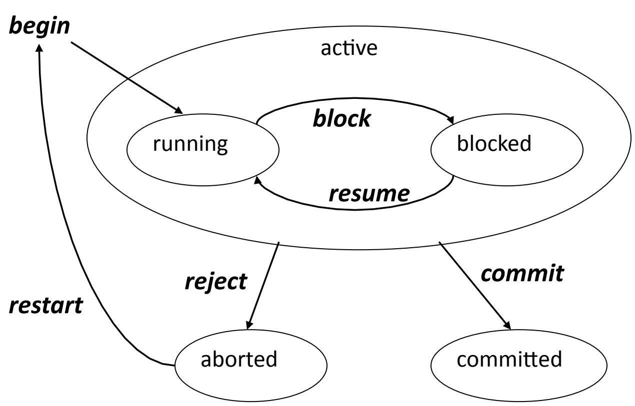
Chapter 4: Concurrency Control Algorithms

- **4.2 General Scheduler Design**
- 4.3 Locking Schedulers
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned

Transaction Scheduler

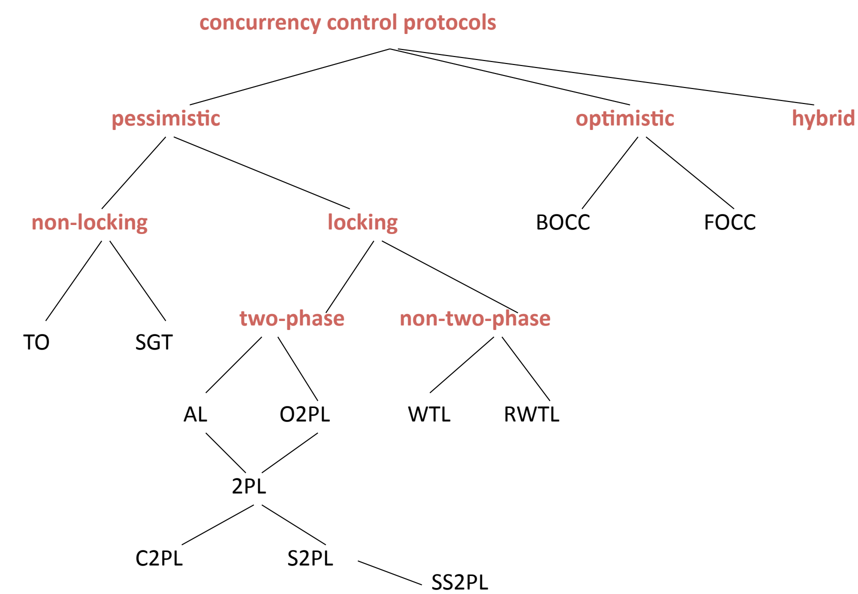


Scheduler Actions and Transaction States



Definition 4.1 (CSR Safety):
For a scheduler S, **Gen(S)** denotes the set of all schedules that S can generate. A scheduler is called **CSR safe** if Gen(S) $\subseteq$ CSR.

Scheduler Classification



Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- **4.3 Locking Schedulers**
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - 4.3.3 Deadlock Handling
  - 4.3.4 Variants of 2PL
  - 4.3.5 Ordered Sharing of Locks (O2PL)
  - 4.3.6 Altruistic Locking (AL)
  - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned

General Locking Rules

For each step the scheduler **requests a lock** on behalf of the step's transaction.
Each lock is requested in a specific **mode (read or write)**.
If the data item is not yet locked in an **incompatible mode** the lock is granted;
otherwise there is a **lock conflict** and the transaction becomes **blocked**
(suffers a **lock wait**) until the current lock holder **releases the lock**.

Compatibility of locks:

lock holder \ lock requestor	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	–
$wl_i(x)$	–	–

General locking rules:

LR1: Each data operation $o_i(x)$ must be preceded by $ol_i(x)$ and followed by $ou_i(x)$.
LR2: For each x and t_i there is at most one $ol_i(x)$ and at most one $ou_i(x)$.
LR3: No $ol_i(x)$ or $ou_i(x)$ is redundant.
LR4: If x is locked by both t_i and t_j, then these locks are compatible.

Simple Locking

- Locking alone is not enough:

$r_1[x]w_2[x]w_2[y]r_1[y]$

$rl_1[x]r_1[x]ru_1[x]wl_2[x,y]w_2[x]w_2[y]wu_2[x,y]rl_1[y]r_1[y]ru_1[y]$

Chapter 4: Concurrency Control Algorithms

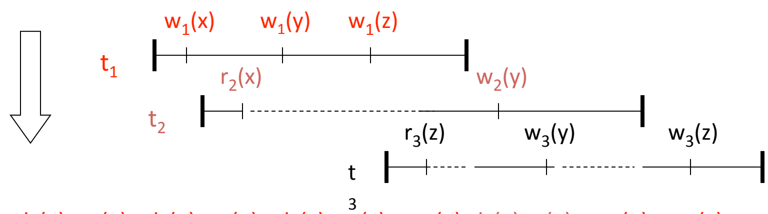
Two Phase Locking Protocol

- The 2PL protocol:
  1. On $p_i[x]$, if $pl_i[x]$ conflicts delay it otherwise set $pl_i[x]$.
  2. Once the scheduler has set $pl_i[x]$ it may not release it until the DM has acknowledged processing of $p_i[x]$.
  3. Once the scheduler has released a lock for a transaction, it may not subsequently obtain any more locks for that transaction (on any data item).

Two-Phase Locking (2PL)

Definition 4.2 (2PL):
A locking protocol is **two-phase (2PL)** if for every output schedule s and every transaction $t_i \in trans(s)$ no ql_i step follows the first ou_i step ($q, o \in \{r, w\}$).

Example 4.4:
s = $w_1(x)$ $r_2(x)$ $w_1(y)$ $w_1(z)$ $r_3(z)$ c_1 $w_2(y)$ $w_3(y)$ c_2 $w_3(z)$ c_3



$wl_1(x)$ $w_1(x)$ $wl_1(y)$ $w_1(y)$ $wl_1(z)$ $w_1(z)$ $wu_1(x)$ $rl_2(x)$ $r_2(x)$ $wu_1(y)$ $wu_1(z)$ c_1
$rl_3(z)$ $r_3(z)$ $wl_2(y)$ $w_2(y)$ $wu_2(y)$ $ru_2(x)$ c_2
$wl_3(y)$ $w_3(y)$ $wl_3(z)$ $w_3(z)$ $wu_3(z)$ $wu_3(y)$ c_3

2PL Properties

- Prop I. If pi[x] in H (which is 2PL) then pli[x] < pi[x] < pui[x].
- Prop II. If conflicting pi[x] and qj[x] in H then either pui[x] < qlj[x] or quj[x] < pli[x].
- Prop III. If pi[x] and qi[y] in H then pli[x] < qui[y].

2PL History is CSR

- Lemma 1. If Ti → Tj in SG(H) then for some x and some conflicting operations pi[x] and qj[x] in H, pui[x] < qlj[x].

- Lemma 2. If T1 → T2 → …→Tn be a path in SG(H), then there exist items x and y such that p1[x] and qn[y] in H such that pu1[x] < qln[y].

Using the Serializability Theorem

- Suppose SG(H) has a cycle: T1→T2→ … →Tn→T1.

Establish contradiction by using Lemma 2.

Proof of 2PL Correctness

Let s be the output of a 2PL scheduler, and let G be the conflict graph of CP (DT(s)) where DT is the projection onto data and termination operations and CP is the committed projection.
The following holds (Lemma 4.2):

(i) If (t_i, t_j) is an edge in G, then $pu_i(x) < ql_j(x)$ for some x with conflicting p, q.
(ii) If $(t_1, t_2, \ldots, t_n)$ is a path in G, then $pu_1(x) < ql_n(y)$ for some x, y.
(iii) G is acyclic.

This can be shown as follows:

(i) By locking rules LR1 through LR4.
(ii) By induction on n.
(iii) Assume G has a cycle of the form $(t_1, t_2, \ldots, t_n, t_1)$.
By (ii), $pu_1(x) < ql_1(y)$ for some x, y,
which contradicts the two-phase property.

Correctness and Properties of 2PL

Theorem 4.1:
Gen(2PL) ⊂ CSR (i.e., 2PL is CSR-safe).

Example 4.5:
$s = w_1(x)\ r_2(x)\ c_2\ r_3(y)\ c_3\ w_1(y)\ c_1 \in CSR$
but $\notin$ Gen(2PL) for $wu_1(x) < rl_2(x)$ and $ru_3(y) < wl_1(y)$,
$rl_2(x) < r_2(x)$ and $r_3(y) < ru_3(y)$, and $r_2(x) < r_3(y)$
would imply $wu_1(x) < wl_1(y)$ which contradicts the two-phase property.

Theorem 4.2:
Gen(2PL) ⊂ OCSR

Example:
$w_1(x)\ r_2(x)\ r_3(y)\ r_2(z)\ w_1(y)\ c_3\ c_1\ c_2$

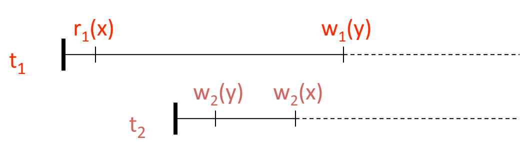
Chapter 4: Concurrency Control Algorithms

- 4.2 General Scheduler Design
- **4.3 Locking Schedulers**
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - **4.3.3 Deadlock Handling**
  - 4.3.4 Variants of 2PL
  - 4.3.5 Ordered Sharing of Locks (O2PL)
  - 4.3.6 Altruistic Locking (AL)
  - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned

Deadlock Detection

Deadlocks are caused by cyclic lock waits
(e.g., in conjunction with lock conversions).

Example:



Deadlock detection:

(i) Maintain dynamic **waits-for graph (WFG)** with active transactions as nodes and an edge from t_i to t_j if t_j waits for a lock held by t_i.

(ii) Test WFG for cycles
- continuously (i.e., upon each lock wait) or
- periodically.

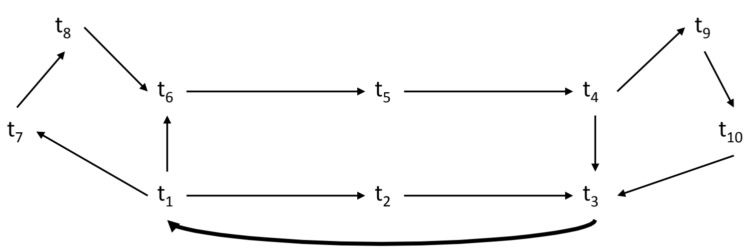
Deadlock Resolution

Choose a transaction on a WFG cycle as a **deadlock victim**
and abort this transaction,
and repeat until no more cycles.

Possible victim selection strategies:
1. Last blocked
2. Random
3. Youngest
4. Minimum locks
5. Minimum work
6. Most cycles
7. Most edges

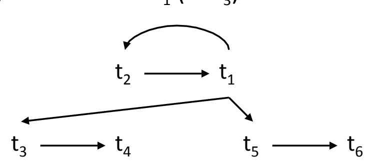
Illustration of Victim Selection Strategies

Example WFG:



Most-cycles strategy would select t_1 (or t_3) to break all 5 cycles.

Example WFG:



Most-edges strategy would select t_1 to remove 4 edges.

Deadlock Prevention

Restrict lock waits to ensure **acyclic WFG** at all times.

Reasonable deadlock prevention strategies:

1. **Wait-die:**
   upon t_i blocked by t_j:
   if t_i started before t_j then wait else abort t_i
2. **Wound-wait:**
   upon t_i blocked by t_j:
   if t_i started before t_j then abort t_j else wait
3. **Immediate restart:**
   upon t_i blocked by t_j: abort t_i
4. **Running priority:**
   upon t_i blocked by t_j:
   if t_j is itself blocked then abort t_j else wait
5. **Timeout:**
   abort waiting transaction when a timer expires

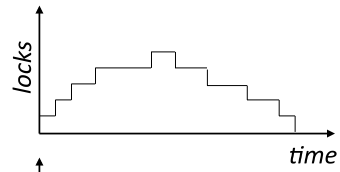
Abort entails later restart.

Chapter 4: Concurrency Control Algorithms

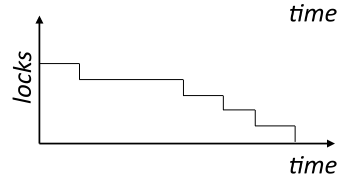
- 4.2 General Scheduler Design
- **4.3 Locking Schedulers**
  - 4.3.1 Introduction
  - 4.3.2 Two-Phase Locking (2PL)
  - 4.3.3 Deadlock Handling
  - **4.3.4 Variants of 2PL**
  - 4.3.5 Ordered Sharing of Locks (O2PL)
  - 4.3.6 Altruistic Locking (AL)
  - 4.3.7 Non-Two-Phase Locking (WTL, RWTL)
  - 4.3.8 Geometry of Locking
- 4.4 Non-Locking Schedulers
- 4.5 Hybrid Protocols
- 4.6 Lessons Learned

Variants of 2PL

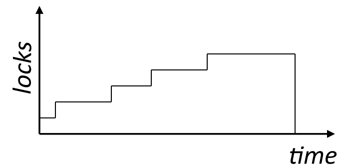
general 2PL



Definition 4.3 (Conservative 2PL):
Under **static** or **conservative 2PL (C2PL)** each transaction acquires all its locks before the first data operation (**preclaiming**).



Definition 4.4 (Strict 2PL):
Under **strict 2PL (S2PL)** each transaction holds all its write locks until the transaction terminates.



Definition 4.5 (Strong 2PL):
Under **strong 2PL (SS2PL)** each transaction holds all its locks (i.e., both r and w) until the transaction terminates.

Properties of S2PL and SS2PL

Theorem 4.3:
Gen(SS2PL) $\subset$ Gen(S2PL) $\subset$ Gen(2PL)

Theorem 4.4:
Gen(SS2PL) $\subset$ COCSR

Chapter 4: Concurrency Control Algorithms

Ordered Sharing of Locks

Motivation:

Example 4.6:

$s_1 = w_1(x)\ r_2(x)\ r_3(y)\ c_3\ w_1(y)\ c_1\ w_2(z)\ c_2$

$\in$COCSR, but
$\notin$Gen(2PL)

Observation:

the schedule were feasible if **write locks could be shared**
s.t. the order of lock acquisitions dictates the order of data operations

Notation:

$\mathbf{pl_i(x) \rightarrow ql_j(x)}$ (with $i \neq j$) for $\mathbf{pl_i(x) <_s ql_j(x) \wedge p_i(x) <_s q_j(x)}$

Example reconsidered with ordered sharing of locks:

$wl_1(x)\ w_1(x)\ rl_2(x)\ r_2(x)\ rl_3(y)\ r_3(y)\ ru_3(y)\ c_3$
$wl_1(y)\ w_1(y)\ wu_1(x)\ wu_1(y)\ c_1\ wl_2(z)\ w_2(z)\ ru_2(x)\ wu_2(z)\ c_2$

Lock Compatibility Tables With Ordered Sharing

LT_1	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	–
$wl_i(x)$	–	–

LT_2	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	→
$wl_i(x)$	–	–

LT_3	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	–
$wl_i(x)$	→	–

LT_4	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	–
$wl_i(x)$	–	→

LT_5	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	→
$wl_i(x)$	→	–

LT_6	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	–
$wl_i(x)$	→	→

LT_7	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	→
$wl_i(x)$	–	→

LT_8	$rl_j(x)$	$wl_j(x)$
$rl_i(x)$	+	→
$wl_i(x)$	→	→

Additional Locking Rules for O2PL

OS1 (lock acquisition):
Assuming that $pl_i(x) \rightarrow ql_j(x)$ is permitted,
if $pl_i(x) <_s ql_j(x)$ then $p_i(x) <_s q_j(x)$ must hold.

Example:
$wl_1(x)\ w_1(x)\ wl_2(x)\ w_2(x)\ wl_2(y)\ w_2(y)\ wu_2(x)\ wu_2(y)\ c_2$
$wl_1(y)\ w_1(y)\ wu_1(x)\ wu_1(y)\ c_1$

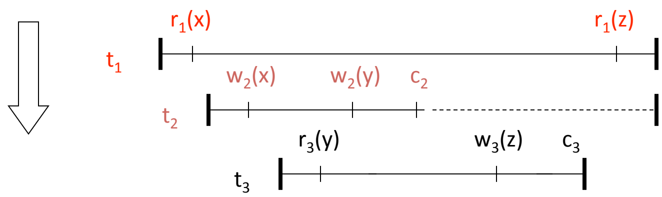
Satisfies OS1,
LR1 – LR4,
is two-phase,
but $\notin$CSR

OS2 (lock release):
If $pl_i(x) \rightarrow ql_j(x)$ and t_i has not yet released any lock, then t_j is **order-dependent** on t_i. If such t_i exists, then t_j is **on hold**.
While a transaction is on hold, it must not release any locks.

O2PL: locking with rules LR1 - LR4, two-phase property, rules OS1 - OS2, and lock table LT_8

O2PL Example

Example 4.7:
s = $r_1(x)$ $w_2(x)$ $r_3(y)$ $w_2(y)$ c_2 $w_3(z)$ c_3 $r_1(z)$ c_1



$rl_1(x)$ $r_1(x)$ $wl_2(x)$ $w_2(x)$ $rl_3(y)$ $r_3(y)$ $wl_2(y)$ $w_2(y)$
$wl_3(z)$ $w_3(z)$ $ru_3(y)$ $wu_3(z)$ c_3 $rl_1(z)$ $r_1(z)$ $ru_1(x)$ $ru_1(z)$ $wu_2(x)$ $wu_2(y)$ c_2 c_1

Correctness and Properties of O2PL

Theorem 4.5:
Let LT_i denote the locking protocol with ordered sharing according to lock compatibility table LT_i.
For each i, $1 \le i \le 8$, Gen(LT_i) $\subseteq$ CSR.

Theorem 4.6:
Gen(O2PL) $\subseteq$ OCSR

Theorem 4.7:
OCSR $\subseteq$ Gen(O2PL)

Corollary 4.1:
Gen(O2PL) = OCSR

Chapter 4: Concurrency Control Algorithms

Altruistic Locking (AL)

Motivation:

Example 4.8: concurrent executions of

$t_1 = w_1(a)\ w_1(b)\ w_1(c)\ w_1(d)\ w_1(e)\ w_1(f)\ w_1(g)$

$t_2 = r_2(a)\ r_2(b)$

$t_3 = r_3(c)\ r_3(e)$

Observations:

- t_2 and t_3 access subsets of the data items accessed by t_1
- t_1 knows when it is "finished" with a data item
- t_1 could "pass over" locks on specific data items to transactions that access only data items that t_1 is finished with (such transactions are "in the wake" of t_1)

Notation:

$d_i(x)$ for t_i **donating** its lock on x to other transactions

Example with donation of locks:

$wl_1(a)\ w_1(a)\ d_1(a)\ rl_2(a)\ r_2(a)\ wl_1(b)\ w_1(b)\ d_1(b)\ rl_2(b)\ r_2(b)\ wl_1(c)\ w_1(c)$ ...

... $ru_2(a)\ ru_2(b)$ ... $wu_1(a)\ wu_1(b)\ wu_1(c)$ ...

Additional Locking Rules for AL

AL1: Once t_i has donated a lock on x, it can no longer access x.

AL2: After t_i has donated a lock x, t_i must eventually unlock x.

AL3: t_i and t_j can simultaneously hold conflicting locks only if t_i has donated its lock on x.

Definition 4.27:

(i) $p_j(x)$ is **in the wake** of t_i ($i \neq j$) in s if $d_i(x) <_s p_j(x) <_s ou_i(x)$.

(ii) t_j is in the wake of t_i if some operation of t_j is in the wake of t_i.
t_j is **completely in the wake** of t_i if all its operations are in the wake of t_i.

(iii) t_j is **indebted** to t_i in s if there are steps $o_i(x)$, $d_i(x)$, $p_j(x)$ s.t.
$p_j(x)$ is in the wake of t_i and ($p_j(x)$ and $o_i(x)$ are in conflict or
there is $q_k(x)$ conflicting with both $p_j(x)$ and $o_i(x)$ and $o_i(x) <_s q_k(x) <_s p_j(x)$.

AL4: When t_j is indebted to t_i,
t_j must remain completely in the wake of t_i.

AL: locking with rules LR1 - LR4, two-phase property, donations, and rules AL1 - AL4 .

AL Example

Example:
$rl_1(a)$ $r_1(a)$ $d_1(a)$ $wl_3(a)$ $w_3(a)$ $wu_3(a)$ c_3
$rl_2(a)$ $r_2(a)$ $wl_2(b)$ $ru_2(a)$ $w_2(b)$ $wu_2(b)$ c_2 $rl_1(b)$ $r_1(b)$ $ru_1(a)$ $ru_1(b)$ c_1

→ disallowed by AL (even ∉CSR)

Example corrected using rules AL1 - AL4:
$rl_1(a)$ $r_1(a)$ $d_1(a)$ $wl_3(a)$ $w_3(a)$ $wu_3(a)$ c_3
$rl_2(a)$ $r_2(a)$ $rl_1(b)$ $r_1(b)$ $ru_1(a)$ $ru_1(b)$ c_1 $wl_2(b)$ $ru_2(a)$ $w_2(b)$ $wu_2(b)$ c_2

→ admitted by AL (t_2 stays completely in the wake of t_1)

Correctness and Properties of AL

Theorem 4.8:
Gen(2PL) $\subset$ Gen(AL).

Theorem 4.9:
Gen(AL) $\subset$ CSR

Example:
s = $r_1(x)$ $r_2(z)$ $r_3(z)$ $w_2(x)$ c_2 $w_3(y)$ c_3 $r_1(y)$ $r_1(z)$ c_1 $\rightarrow \in$ CSR, but $\notin$ Gen(AL)

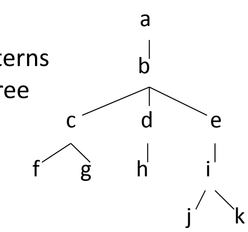
Chapter 4: Concurrency Control Algorithms

(Write-only) Tree Locking

Motivating example:

concurrent executions of transactions with access patterns that comply with organizing data items into a virtual tree

$t_1 = w_1(a)\ w_1(b)\ w_1(d)\ w_1(e)\ w_1(i)\ w_1(k)$
$t_2 = w_2(a)\ w_2(b)\ w_2(c)\ w_2(d)\ w_2(h)$



Definition (Write-only Tree Locking (WTL)):
Under the **write-only tree locking protocol (WTL)** lock requests and releases must obey LR1 - LR4 and the following additional rules:

WTL1: A lock on a node x other than the tree root can be acquired only if the transaction already holds a lock on the parent of x.

WTL2: After a $wu_i(x)$ no further $wl_i(x)$ is allowed (on the same x).

Example:
$wl_1(a)\ w_1(a)\ wl_1(b)\ wu_1(a)\ w_1(b)\ wl_2(a)\ w_2(a)\ wl_1(d)\ w_1(d)\ wu_1(d)\ wl_1(e)\ wu_1(b)\ w_1(e)\ wl_2(b)\ wu_2(a)\ w_2(b)$ ...

Correctness and Properties of WTL

Lemma 4.6:
If t_i locks x before t_j does in schedule s, then for each successor v of x that is locked by both t_i and t_j the following holds: $wl_i(v) <_s wu_i(v) <_s wl_j(v)$.

Theorem 4.10:
Gen(WTL) $\subseteq$ CSR.

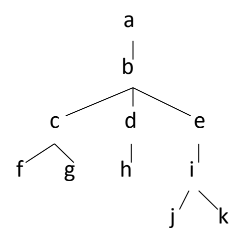
Theorem 4.11:
WTL is deadlock-free.

Comment: WTL is applicable even if a transaction's access patterns are not tree-compliant, but then locks must still be obtained along all relevant paths in the tree using the WTL rules.

Read-Write Tree Locking

Problem: t_i locks root before t_j does,
but t_j passes t_i within a "read zone"

Example:
$rl_1(a)\ rl_1(b)\ r_1(a)\ r_1(b)\ wl_1(a)\ \mathbf{w_1(a)}\ wl_1(b)\ ul_1(a)\ rl_2(a)\ \mathbf{r_2(a)}$
$w_1(b)\ rl_1(e)\ ul_1(b)\ rl_2(b)\ r_2(b)\ ul_2(a)\ rl_2(e)\ rl_2(i)\ ul_2(b)\ r_2(e)\ \mathbf{r_1(e)}$
$r_2(i)\ wl_2(i)\ \mathbf{w_2(i)}\ wl_2(k)\ ul_2(e)\ ul_2(i)\ rl_1(i)\ ul_1(e)\ \mathbf{r_1(i)}$ ...



$\rightarrow$ appears to follow TL rules
but $\notin$ CSR

Solution: formalize "read zone"
and enforce two-phase property on "read zones"

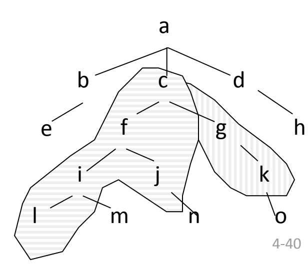
4/18/11 Transactional Information Systems 4-39

Locking Rules of RWTL

For transaction t with read set RS(t) and write set WS(t)
let $C_1, \ldots, C_m$ be the connected components of RS(t).
A **pitfall** of t is a set of the form
$C_i \cup \{x \in WS(t) \mid x \text{ is a child or parent of some } y \in C_i\}$.

Definition (read-write tree locking (RWTL)):
Under the **read-write tree locking protocol (RWTL)** lock requests and releases
Must obey LR1 - LR4, WTL1, WTL2, and the two-phase property within each pitfall.

Example:
t with RS(t)={f, i, g} and WS(t)={c, l, j, k, o}
has pitfalls pf_1={c, f, i, l, j} and pf_2={g, c, k}.



4/18/11 Transactional Information Systems 4-40

Correctness and Generalization of RWTL

Theorem 4.12:
Gen (RWTL) $\subseteq$ CSR.

RWTL can be generalized for a DAG organization of data items into a **DAG locking** protocol with the following additional rule:
t_i is allowed to lock data item x only if holds locks on a majority of the predecessors of x.